EDAF75
Database Technology

Lecture 10

Feb 20, 2025

Today

- Quick intro to the project
- Examples from last time (triggers)
- SQL injection
- Indexes and query planning
- Limitations of SQL
- Stored procedures
- Alternatives to SQL (NoSQL)
- Scaling

Administation

- On Monday, Neo Technologies will give a guest lecture, presenting their graph database: Neo4j – it's a very interesting alternative to the SQL databases we've seen so far. If you bring your laptop, you can try out some examples they'll show.
- The problem text for the project was announced last week, next week we'll have design meetings (you'll have to sign up for it just as you have signed up for the labs).
- We'll assign a project supervisor to each group – you'll get an email with more details later.

Project

- Goal: *Design a database for a small bakery, and implement a REST service using the database as a backend*
- Create an ER-model, and draw it as a UML class diagram (so, no "Crows foot", "Chen", or other notations) – this diagram should be posted to a private coursegit repository which you will share with your project supervisor
- We will try to work out some format for design discussions between groups – the exact format hasn't been decided yet
- I encourage you to discuss your design with other groups in whatever settings you can come up with on your own
- After the design meeting, when your design has been approved, we'll release a REST API which you should implement

Project

- When you have something which needs approval, push your code and report to `coursegit.cs.lth.se`, and notify your supervisor
- The supervisors have two roles:
  - to make sure your design is OK before you start coding
  - to make sure your program is OK when you finish
- The project is a part of the examination, and the supervisors are not part of the groups, so *they're not supposed to help out with your troubleshooting*
- Each project supervisor will have *many* project to keep track of, and they will at the same time be busy with other courses, so unfortunately the feedback may sometimes take several days
- Deadline: you must have pushed your final (approvable) version to `coursegit.cs.lth.se` no later than 23:59 on April 30

Project

- Some of you have programmed a lot before taking this course, many of you haven't – we'll require that all of you do your best (but no more), *and that you follow common coding guidelines*, such as:
  - *The specs must be followed exactly*
  - Never, ever, use tabs in your Java or Python code!
  - Always use correct indentation (4 spaces in Python, and 2 or 4 spaces in), and place braces where the standard guidelines put them
  - Functions/methods should generally do only one thing each – a function/method which does many different things should be broken into several functions/methods
  - Use proper names – functions/methods/parameters should have descriptive names, local variables can be shortened (i.e., the size of its scope determines how descriptive a name must be)

Transactions and triggers

- Last time, we talked about transactions and triggers
- The project will be much easier to implement if you use transactions and triggers properly, and we will require you to use them appropriately

Triggers

Example

Experiment some with triggers, foreign keys, and Python exceptions

Python and SQLite

- The `PRAGMA foreign_key` switch only works for the current connection, we need to turn it on each time our Python program runs
- Rollbacks and constraint violations in the database will be returned as some kind of `sqlite3.Error`

Removal of foreign keys

- Normally we're not allowed to remove a row with a foreign key which is referenced from another row (probably in another table)
- We can use a '`foreign-key-clause`' (in the table definition) to handle a removal which could be problematic

Foreign key clauses

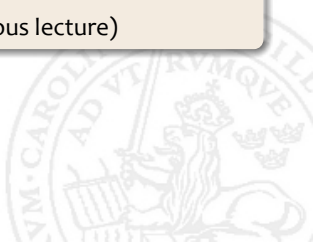
Example

Implement a foreign key clause which removes all applications by a student which is removed from the `students` table

Example

Example

- Write a trigger to make sure no student applies for a major which hasn't been seen before on the college in question.
- Why is this a weird problem?
- How do we solve it? (It is a problem in the notes for previous lecture)



Indicies

- When we define a primary key, the database creates a special *index* to make searches for the key fast – indexes can also speed up joins and `ORDER BY` statements
- We can create our own indexes, with as many columns as we want

```
CREATE INDEX names_and_ages
ON employees(last_name, first_name, birth_year);
```

- This index will speed up searches for:
  - last name
  - last name and then first name
  - last name and birth year (with some fiddling)
- The index above will not help much if we're just searching for first name, or birth year
- Indicies make some things faster, but insertions and deletions will become slower
- We can sometimes create a *covering index*, it is an index which includes the value we're normally looking for – using a covering index the DBMS will not even have to look at the table itself

Indicies and the Query Planner

- For each SQL statement, there might be thousands of ways to perform the operation
- Before a DBMS executes a statement, its *query planner* takes a good look at it, to find a way to execute the statement as efficiently as possible
- Indicies are very important during the planning – when searching for a value which isn't indexed properly, the DBMS might have to do a linear search through a table
- Sometimes big joins (especially cross joins) gives the query planner so many alternatives that the planning itself takes substantial time
- In `SQLite3` there is a command `EXPLAIN QUERY PLAN` which explains what will happen during a given query
- `SQLite3` also has a similarily named command `EXPLAIN`, which shows the VM-instructions it would use for a given query

Example

Example

Write a program which keeps track of the names of your friends, and write code to insert new friends into a SQLite database – see the notebook

SQL Injection

- We must be careful before putting user data into our database – never concatenate parameters into a query!
- SQL injection is when a user sends a string which alters the intended meaning of our SQL statement
- A classic example is the statement

```
stmt = "SELECT * FROM users WHERE name = '"
       + user_name + "';"
```

  and `user_name` gets the value `"' OR '1'='1"`
- Using `PreparedStatement` instead of `Statement` makes our code safer
- In the project, we'll require that you use a `PreparedStatement` where a `Statement` would have been dangerous

SQL injection

- On wikipedia there is a long list of known SQL injections (and we probably don't hear about most of the successful ones)
- So, SQL injection is a real thing, and we'd better be safe than sorry – using a `PreparedStatement` is a very simple protective measure
- In languages such as C/C++, there are relatives to SQL injection – one well known example is "buffer overruns", in which someone crafts a message which overflows the buffer created to hold the reply (this was one of the exploits used by the "Morris worm")
- Very few programmers have regretted writing code which was 'too safe'

Exercise

Exercise

Define a table with employees of a company, and information about their immediate supervisors.

- Write a query which finds the name of the immediate supervisor for each employee
- Write a query which finds the names of the supervisor of the supervisor for each employee
- Write a query which finds all the supervisors (transitively) of an employee

Limitations of SQL

- For a long time, SQL lacked recursion (and loops), and many DBMS's still do
- That means there are many simple things we can't do easily in SQL, such as traversing a varying number of steps in some kind of list-like structure (e.g., a line of ascendants in a tree)
- That shortcoming can be dealt with in several ways:
  - by moving the iterative code to the clients (so we write our recursive calls and loops in another language, like Java or Python)
  - by using *stored procedures*
  - by using a graph database
- Many DMBS's now have some kind of recursion (`SQLite3` is one of them), but using it is somewhat difficult and error prone

Recursion in PostgreSQL/SQLite

- In PostgreSQL and `SQLite3`, as an example, we can 'loop' to create a table with the numbers 1..10:

```
WITH RECURSIVE enumeration(x) AS (
  VALUES(1)
  UNION
  SELECT x+1
  FROM   enumeration
  WHERE  x<10
)
SELECT x FROM enumeration;
```

- A detailed understanding of recursive queries is beyond the scope of the course, but I want you to be aware of them

Computer times adjusted to human scale (2017)

One CPU cycle	0.4 ns	1 s
Level 1 cache access	0.9 ns	2 s
Level 2 cache access	2.8 ns	7 s
Level 3 cache access	28 ns	1 min
Main memory access (DDR DIMM)	100 ns	4 min
Intel Optane DC SSD I/O	<10 µs	7 hrs
NVMe SSD I/O	25 µs	17 hrs
SSD I/O	50-150 µs	1.5-4 days
Rotational disk I/O	1-10 ms	1-9 months
Internet call: San Francisco to New York City	65 ms	5 years
Internet call: San Francisco to Hong Kong	141 ms	11 years

Practical consequences of the timescale

- We try to keep the number of network calls to a minimum – `WITH`-statements, stored procedures and triggers can help us
- We try to minimize the number of disk accesses – DMBS's often do this by using B-trees
- We try to write code which has a memory footprint which is as 'local' as possible (this is not really database related)

Stored procedures

- We've seen that SQL lacks some simple features, like user defined functions, and loops
- Often this can be alleviated by writing code in other languages, and have them call SQL queries/statements 'remotely' (like we've done in Java and Python)
- Some DBMS's have *stored procedures*, which are user defined functions, and in which we can use regular programming features such as parameters and loops
- One of the main advantages with stored procedures is that it reduces the need for calls over a network connection
- The way `SQLite3` operates makes stored procedures almost pointless – there are no network calls, since the database code is linked into our program

Using a stored procedure in MySQL

```
CREATE FUNCTION getTopLevelSupervisorName(IN emp_name VARCHAR(10))
       RETURN VARCHAR(10)
BEGIN
    DECLARE b_id INT;
    DECLARE b_name VARCHAR(10);

    SET b_name = emp_name;
    SELECT supervisor_id into b_id
    FROM employees
    WHERE name = emp_name;
    WHILE b_id <> 0 DO
        SELECT name, supervisor_id
        INTO   b_name, b_id
        FROM   employees
        WHERE  nbr = b_id;
    END WHILE;
    RETURN b_name;
END;
```

Big Data

- The cost of storing 1 GB of data has gone down from ca SEK 1 000 000 in the 1980-ies, to less than SEK 1 today
- Today, there are many databases on the petabyte scale (that is 1 000 000 000 000 000 bytes)
- Data mining is a very active field today, and we mine for:
  - Scientific discoveries (CERN invented the web for this purpose)
  - User behavior (customize advertising)
  - Economic data (algorithmic trading)
  - …
- To find interesting patters in our data, we often want to store many kinds of data:
  - text
  - urls
  - images
  - sound
  - video

OLTP and OLAP

- **OLTP:** *OnLine Transaction Processing* – is used for real time processing of business data
- **OLAP:** *OnLine Analytical Processing* – is used for making complicated queries on historical data
- *Column store*: a database with rows containing columns of data, but data is organized so that each column is saved for quick access – this makes many queries faster, and is typically used for OLAP

SQL vs. NoSQL

- Relational databases have been a phenomenal success, for several reasons:
  - ACID
  - Well tested technology
  - An enormous amount of money invested in them
- It is not a panacea for all data, though
  - They require that we convert all data into tables
  - They may be more complex than necessary
  - They're sometimes not fast enough
  - They don't scale very well (more about that soon)

NoSQL

- In 2000-2016, many alternatives to relational databases sprung up – for some reason they got the moniker NoSQL (it should really have been NonRelational)
- Initially it meant literally "No SQL", but it has evolved into meaning "Not Only SQL", since many of them embed some SQL like query language themselves
- NoSQL databases allow users to save and access all sorts of data in simple ways
- NoSQL databases often sacrifice some of the ACID properties, sometimes they replace "Consistency" with "Eventual consistency"

NoSQL

- There are different kinds of NoSQL databases:
  - *Key-value store*: this is the simplest kind of database, it's basically a `Map[K,V]`, where the database has no way of querying on the contents – Redis is a popular key-value store
  - *Document store*: this is an enhanced key-value store, where we can search and manipulate data based on the contents, not only the keys – MongoDB and CouchDB are popular document stores
  - *Wide column store*: has tables, rows and columns, but the columns can vary from row to row (Apache Cassandra is a popular example)
  - *Graph databases*: data is stored in nodes and edges in a graph – Neo4j (from Malmö!) is a popular example

NoSQL – MongoDB

Exercise

Use MongoDB to insert some student with their applications, and to make some simple queries.

MongoDB

- We save data as JSON-like objects in 'documents'
- We don't have to define schemas (but can do)
- We can query using a simple query language

MongoDB – examples

```
use db.students

db.students.insertMany([
{
  id: 123,
  name: "Amy",
  gpa: 3.9,
  applications: [
    {college: "Berkeley", major: "CS"},
    {college: "Cornell", major: "EE"},
    {college: "Stanford", major: "CS"},
    {college: "Stanford", major: "EE"}
  ]
},
{
  id: 234,
  name: "Bob",
  gpa: 3.6,
  applications: [
    {college: "Berkeley", major: "biology"}
  ]
}])
```

MongoDB – examples

```
use db.students

db.students.find({})
db.students.find({}).pretty()

db.students.find({"id": 123}).pretty()

db.students.find({"applications.college": "Stanford"}).pretty()
db.students.find({"applications.major": "CS"}).pretty()

db.students.remove({name: "Amy"})

db.students.drop()
```

Scaling

- Handling bigger amounts of data requires that we add computing power and storage space, we can do it in at least two ways:
  - *Vertical scaling*: updating our processor and expanding our memory – this is also called *scaling up*
  - *Horizontal scaling*: adding more processors/computers and more hard drives – this is also called *scaling out*
- Horizontal scaling is normally cheaper, and has greater potential

Capitalizing on horizontal scaling

- If we scale horizontally, we need a way to use our resources in parallel
- Google used *map-reduce* for a long time – a simplistic description:
  1. We distribute our data to many servers
  2. On each server, we 'map' a function to our data
  3. We then 'reduce' the results from each server into one final result
- `map` and `reduce` are staples of functional programming
- Hadoop is an open source framework based on map-reduce

Horizontal scaling of databases

- Databases are sometimes split row-wise into non-overlapping partitions, this is called *horizontal partitioning*
- If the partitions are put on separate servers, we call it *sharding*
- A global company could potentially use sharding to put European customers on European servers, and American customers on American servers – this could make some queries faster
- Horizontal partitioning (and sharding in particular) enables us to use map-reduce-like algorithms
- The cost of sharding is increased complexity, and a reliance on the connections between the servers
- MongoDB and Spanner are examples of DBMS's using sharding